.NET Full Stack Development Program

Day 4 Collections

Outline

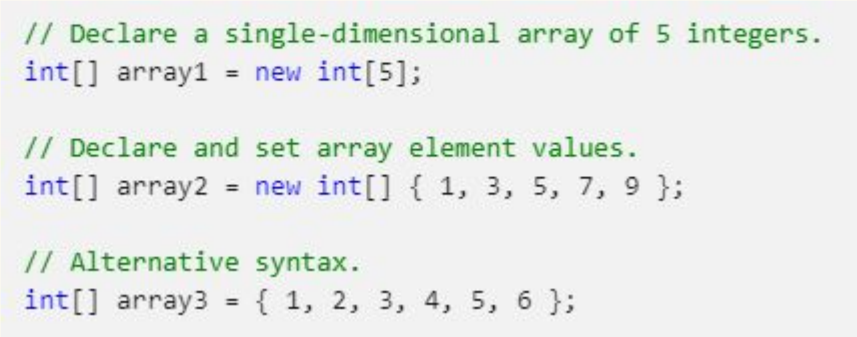
- Array
- Collections
    - Generic Collections
        - List, LinkedList, Dictionary, HashSet, SortedList, Stack, Queue
    - Non-generic Collections
        - ArrayList, HashTable, SortedList, Stack, Queue
- Comparisons and Sorts within Collections

Collection

- Group of objects
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated
- Following constructors are common to all classes implementing Collection
  - T()
  - T(){...}
  - T(Collection c)

Array

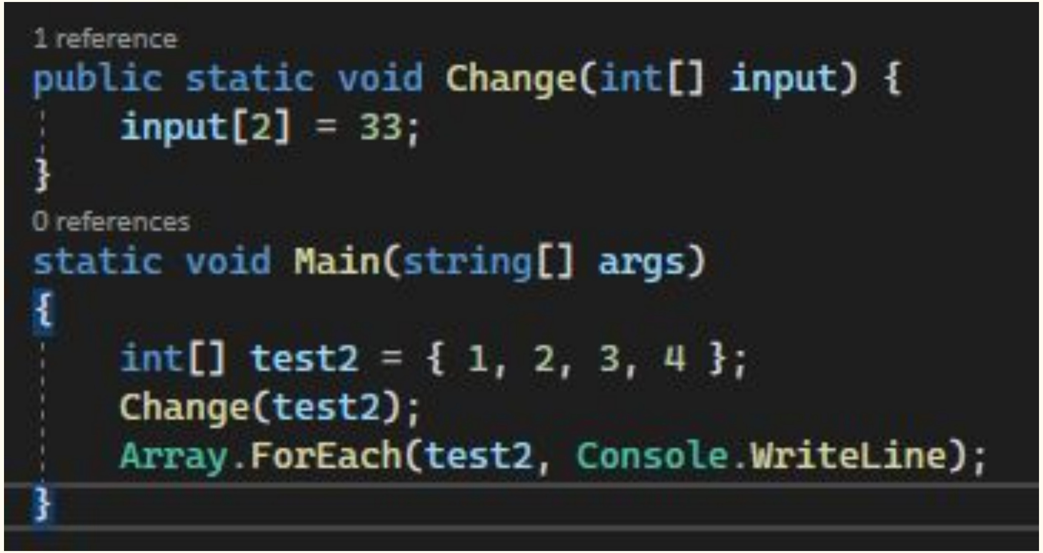
- Can store multiple variables of the **same type**
- Size of the array is **fixed** when the array instance is created
- If you want the array to store elements of any type, you can specify **object** as its type
- The default values of numeric array elements are set to **zero**, and reference elements are set to **null**



```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = new int[] { 1, 3, 5, 7, 9 };

// Alternative syntax.
int[] array3 = { 1, 2, 3, 4, 5, 6 };
```

Array

- Passing Arrays as Argument



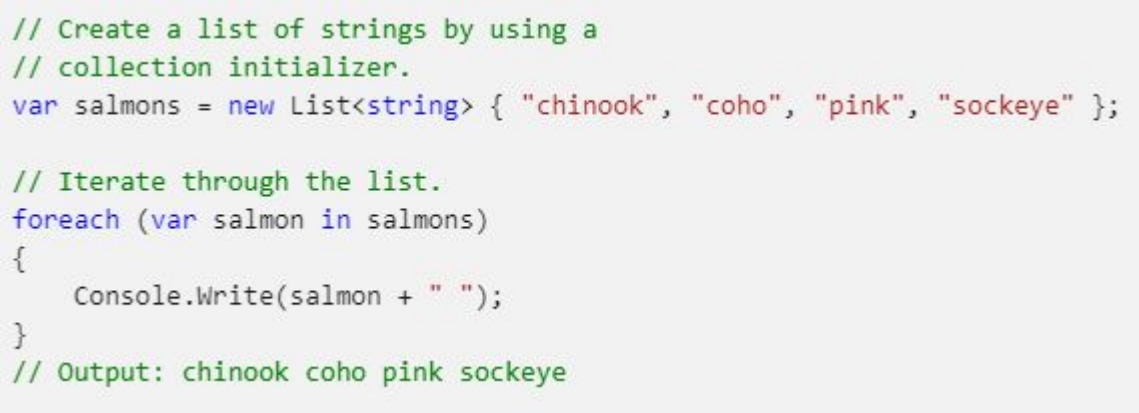
```
1 reference
public static void Change(int[] input) {
    input[2] = 33;
}
0 references
static void Main(string[] args)
{
    int[] test2 = { 1, 2, 3, 4 };
    Change(test2);
    Array.ForEach(test2, Console.WriteLine);
}
```

Collections

- Generic Collections
  - Generic Collections work on the **specific type** that is specified in the program whereas non-generic collections work on the **object** type.
  - Using System.Collections.Generic;
  - List, LinkedList, Dictionary, HashSet, SortedList, Stack, Queue
- Non-generic Collections
  - In non-generic collections, each element can represent a value of a **different type**. The collection size is not fixed. Items from the collection can be added or removed at runtime
  - Using System.Collections;
  - ArrayList, HashTable, SortedList, Stack, Queue

Generic Collection

- List
  - List class is a collection that can be used for **specific types**.
  - List is a class that is similar to an array, but the size is not fixed
  - Elements can be added / removed at runtime.
  - Ex. `List<int> al = new List<int>();`



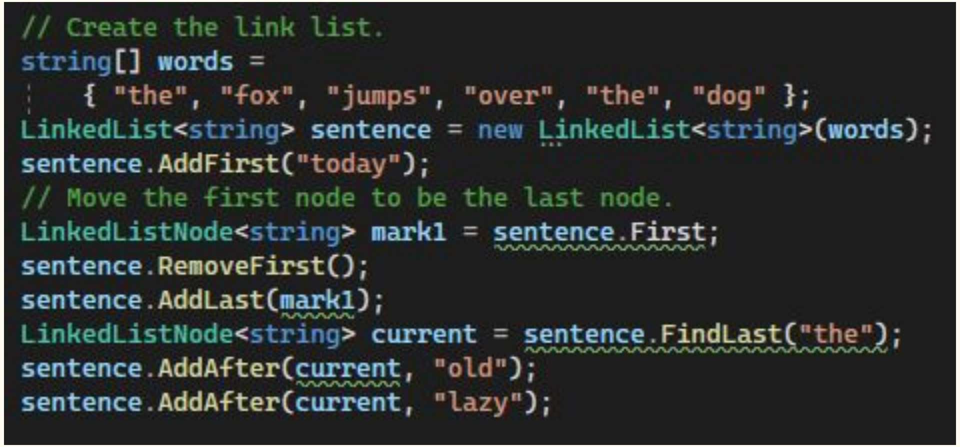
```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Generic Collection

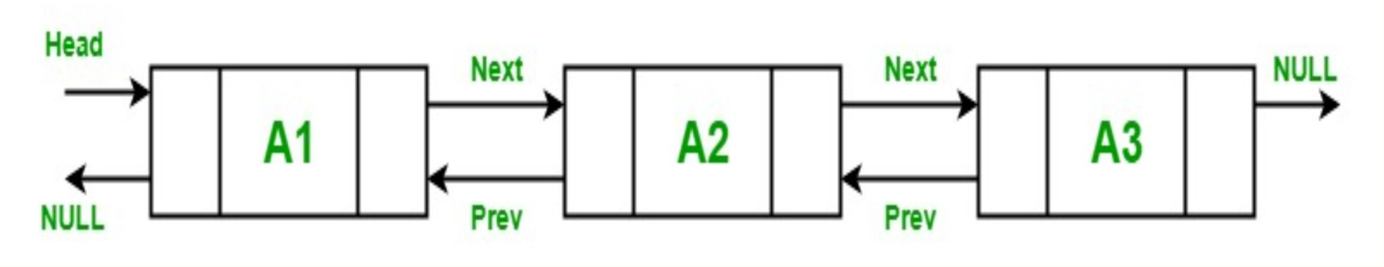
- List
    - access
    - Remove()
    - RemoveAt()
    - Contains()
    - IndexOf()
    - LastIndexOf()

Generic Collection

- LinkedList
  - a general-purpose linked list (doubly linked)
  - LinkedList<T> provides separate nodes of type **LinkedListNode<T>**, so insertion and removal are **O(1)** operations.

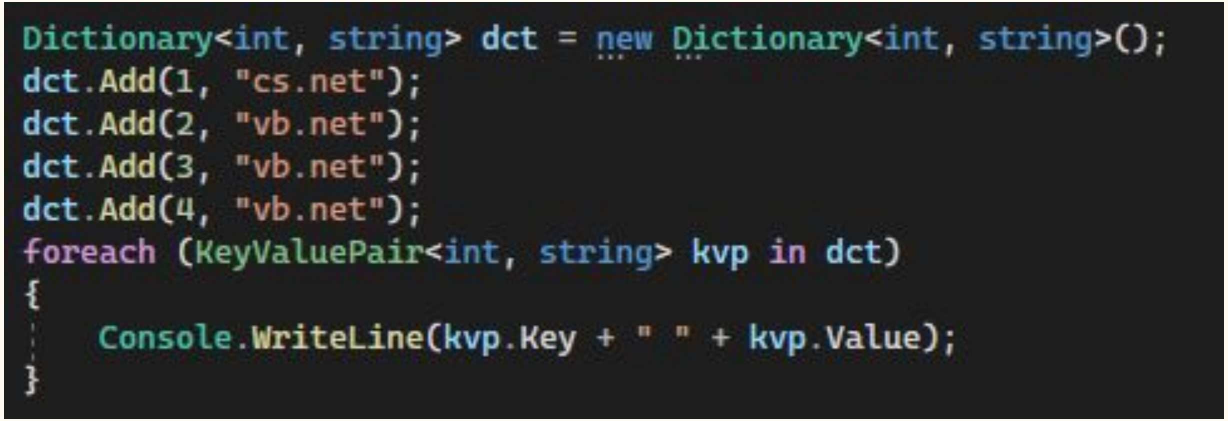


```
// Create the link list.
string[] words =
    { "the", "fox", "jumps", "over", "the", "dog" };
LinkedList<string> sentence = new LinkedList<string>(words);
sentence.AddFirst("today");
// Move the first node to be the last node.
LinkedListNode<string> mark1 = sentence.First;
sentence.RemoveFirst();
sentence.AddLast(mark1);
LinkedListNode<string> current = sentence.FindLast("the");
sentence.AddAfter(current, "old");
sentence.AddAfter(current, "lazy");
```


Head
NULL
A1
Next
Prev
A2
Next
Prev
A3
NULL

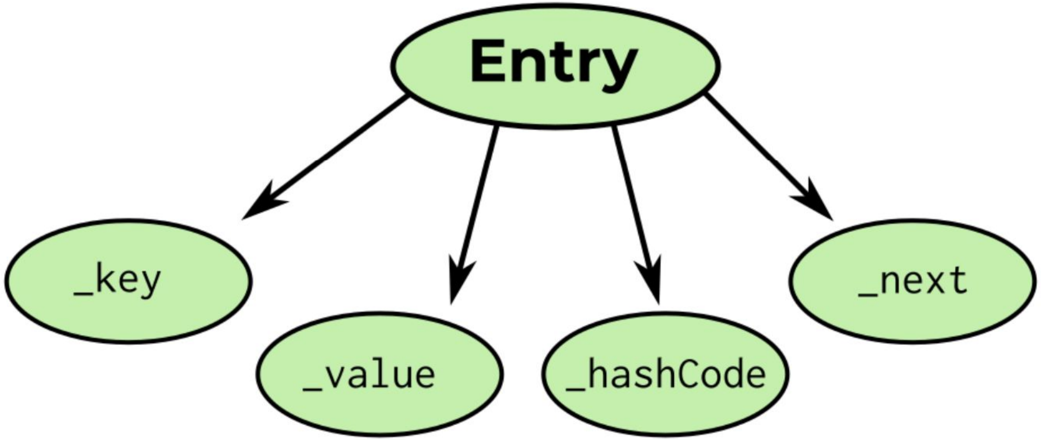
Generic Collection

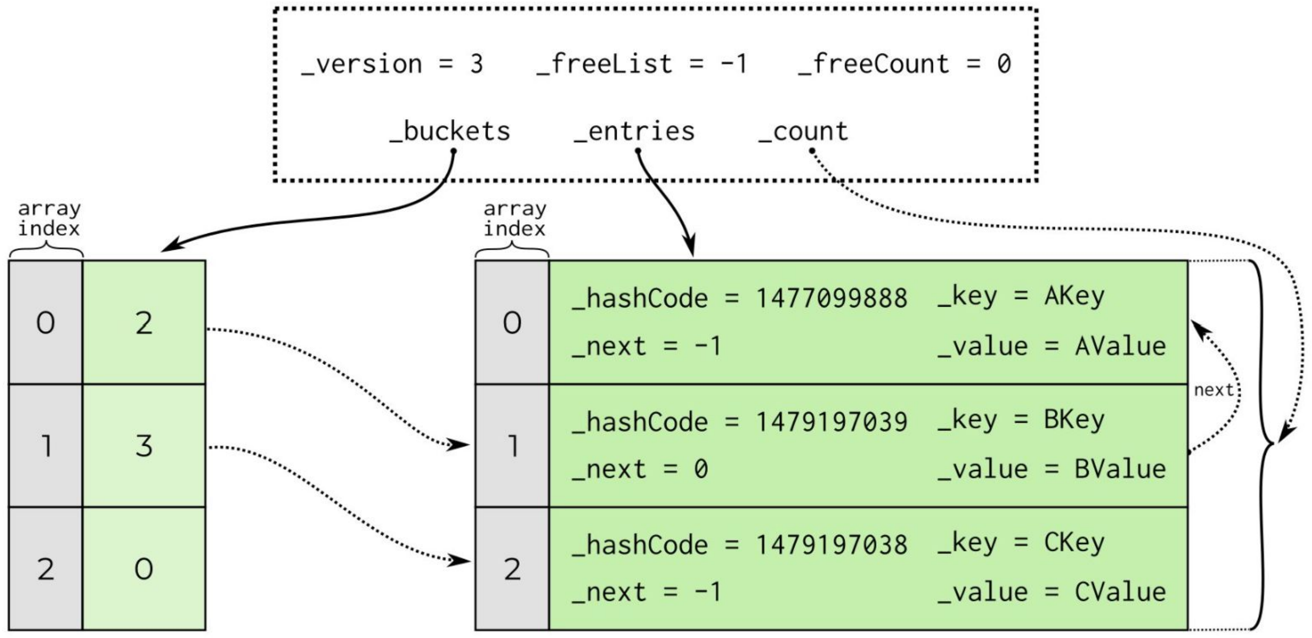
- Dictionary
  - represents the items as a combination of a key and value
  - access the value based on the key



```
Dictionary<int, string> dct = new Dictionary<int, string>();
dct.Add(1, "cs.net");
dct.Add(2, "vb.net");
dct.Add(3, "vb.net");
dct.Add(4, "vb.net");
foreach (KeyValuePair<int, string> kvp in dct)
{
    Console.WriteLine(kvp.Key + " " + kvp.Value);
}
```

Entry struct





_version = 3 _freeList = -1 _freeCount = 0

_buckets _entries _count

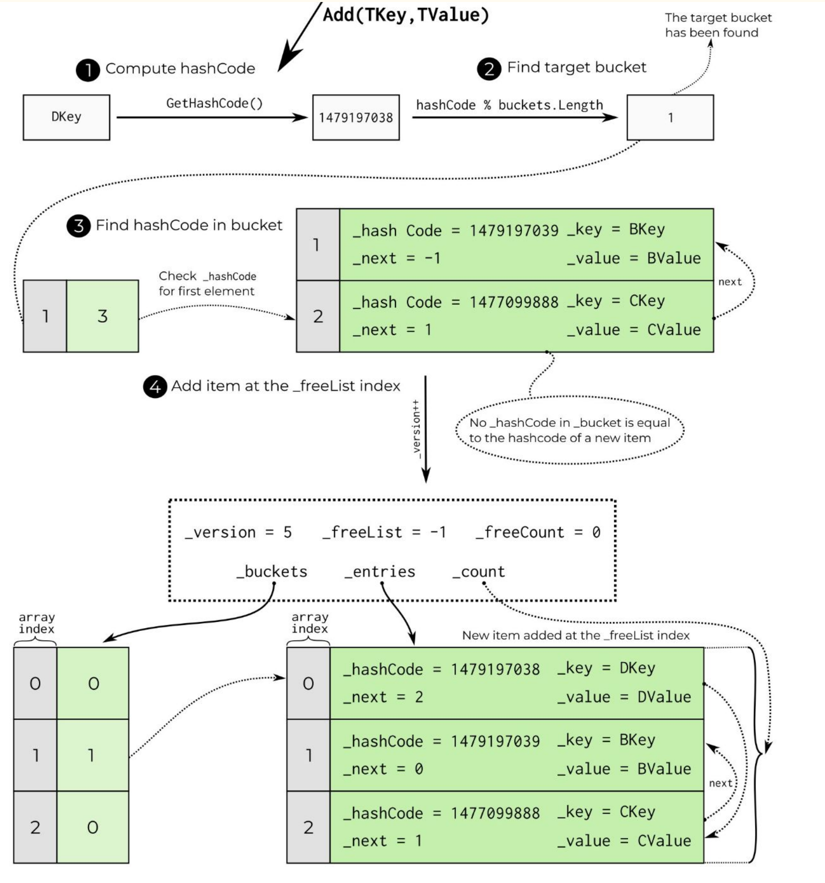
array index	
0	2
1	3
2	0

array index		

0	_hashCode = 1477099888
	_next = -1
		_key = AKey
		_value = AValue

1	_hashCode = 1479197039
	_next = 0
		_key = BKey
		_value = BValue

2	_hashCode = 1479197038
	_next = -1
		_key = CKey
		_value = CValue



Add(TKey,TValue)

1 Compute hashCode

DKey → GetHashCode() → 1479197038

2 Find target bucket

hashCode % buckets.Length → 1

The target bucket has been found

3 Find hashCode in bucket

1	3

Check _hashCode for first element

1	_hash Code = 1479197039
	_next = -1
		_key = BKey
		_value = BValue

2	_hash Code = 1477099888
	_next = 1
		_key = CKey
		_value = CValue

next

No _hashCode in _bucket is equal to the hashcode of a new item

4 Add item at the _freeList index

_version++

```
_version = 5    _freeList = -1    _freeCount = 0

      _buckets     _entries     _count
```

array index

0	0
1	1
2	0

array index

New item added at the _freeList index

0	_hashCode = 1479197038
	_next = 2
		_key = DKey
		_value = DValue

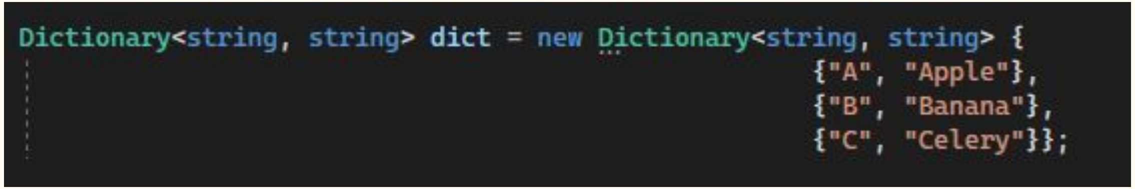
1	_hashCode = 1479197039
	_next = 0
		_key = BKey
		_value = BValue

2	_hashCode = 1477099888
	_next = 1
		_key = CKey
		_value = CValue

next

Generic Collection

- Dictionary
  - Quick initialization

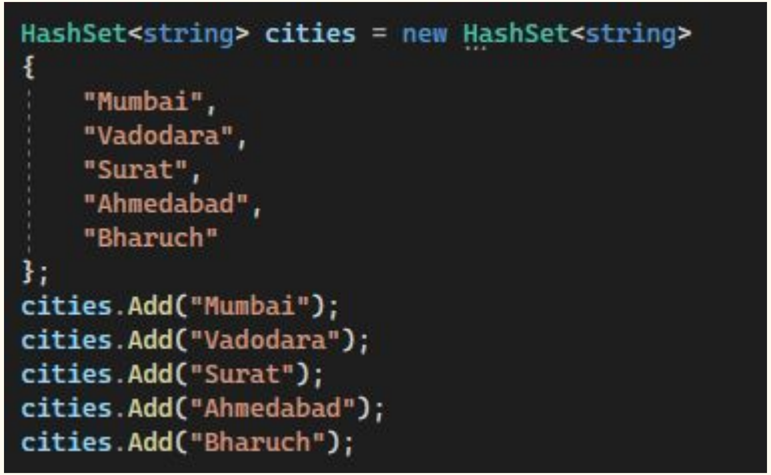


```
Dictionary<string, string> dict = new Dictionary<string, string> {
                                                     {"A", "Apple"},
                                                     {"B", "Banana"},
                                                     {"C", "Celery"}};
```

  - Methods
    - ContainsKey()
    - TryGetValue()
    - TryAdd()

Generic Collection

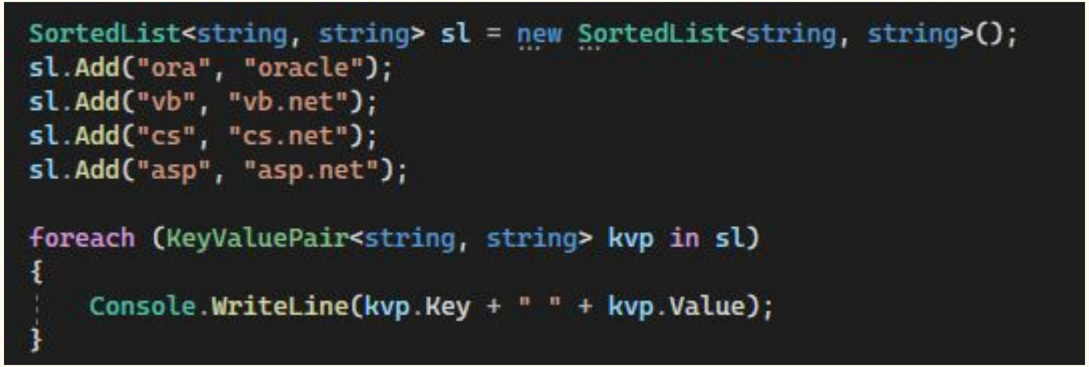
- HashSet
    - a collection that contains no duplicate elements, and whose elements are in no particular order



```
HashSet<string> cities = new HashSet<string>
{
    "Mumbai",
    "Vadodara",
    "Surat",
    "Ahmedabad",
    "Bharuch"
};
cities.Add("Mumbai");
cities.Add("Vadodara");
cities.Add("Surat");
cities.Add("Ahmedabad");
cities.Add("Bharuch");
```

Generic Collection

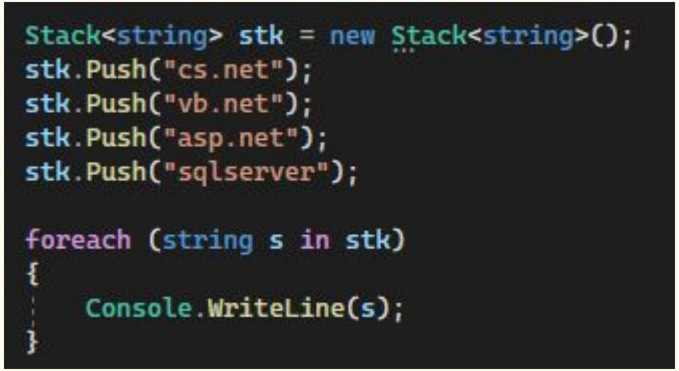
- SortedList
    - represents a collection of key/value pairs that are **sorted by key** based on the associated **IComparer<T>** implementation.



```
SortedList<string, string> sl = new SortedList<string, string>();
sl.Add("ora", "oracle");
sl.Add("vb", "vb.net");
sl.Add("cs", "cs.net");
sl.Add("asp", "asp.net");

foreach (KeyValuePair<string, string> kvp in sl)
{
    Console.WriteLine(kvp.Key + " " + kvp.Value);
}
```

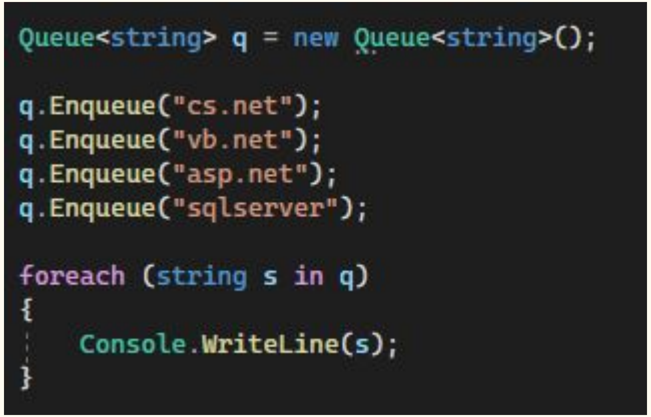
Generic Collection

- Stack
    - It represents a last-in, first out collection of object



```
Stack<string> stk = new Stack<string>();
stk.Push("cs.net");
stk.Push("vb.net");
stk.Push("asp.net");
stk.Push("sqlserver");

foreach (string s in stk)
{
    Console.WriteLine(s);
}
```

- Queue
    - It represents a first-in, first out collection of object



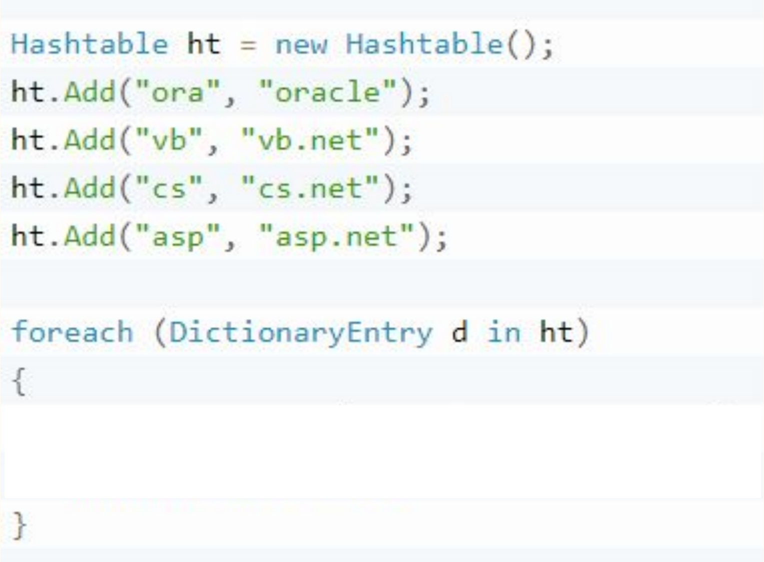
```
Queue<string> q = new Queue<string>();

q.Enqueue("cs.net");
q.Enqueue("vb.net");
q.Enqueue("asp.net");
q.Enqueue("sqlserver");

foreach (string s in q)
{
    Console.WriteLine(s);
}
```

Non-generic Collection

- ArrayList
    - ArrayList class is a collection that can be used for **any types or objects.**
        - Arraylist is a class that is similar to an array, but it can be used to store values of various types.
        - An Arraylist doesn't have a specific size.
        - Any number of elements can be stored.
        - Ex. `ArrayList al = new ArrayList();`

Non-generic Collection

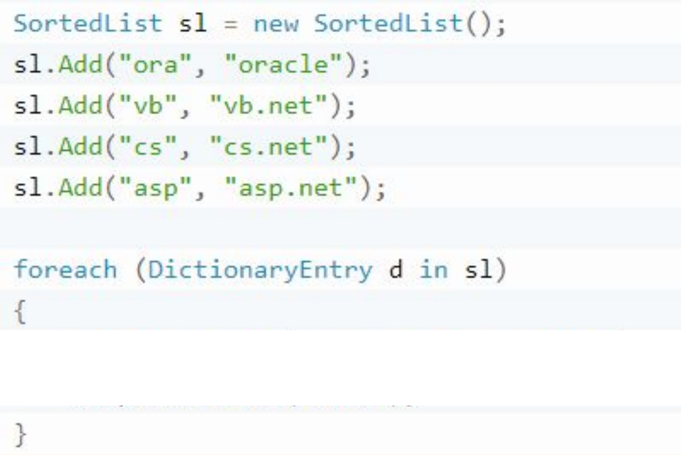
- HashTable
  - represents the items as a combination of a key and value



```
Hashtable ht = new Hashtable();
ht.Add("ora", "oracle");
ht.Add("vb", "vb.net");
ht.Add("cs", "cs.net");
ht.Add("asp", "asp.net");

foreach (DictionaryEntry d in ht)
{


}
```

Non-generic Collection

- SortedList
    - is a class that has the combination of arraylist and hashtable.
    - represents a collection of key/value pairs that are **sorted by key** and are accessible by key and by index



```
SortedList sl = new SortedList();
sl.Add("ora", "oracle");
sl.Add("vb", "vb.net");
sl.Add("cs", "cs.net");
sl.Add("asp", "asp.net");

foreach (DictionaryEntry d in sl)
{


}
```

Non-generic Collection

- Stack
  - It represents a last-in, first out collection of object
  - It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item
- Queue
  - It represents a first-in, first out collection of object
  - It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called enqueue and when you remove it, it is called deque

Comparison and Sorts within Collections

Comparison (check for equality)

- If type T implements the **IEquatable<T>** generic interface, then the equality comparer is the **Equals** method of that interface.

- If type T does **not** implement **IEquatable<T>**, **Object.Equals** is used.

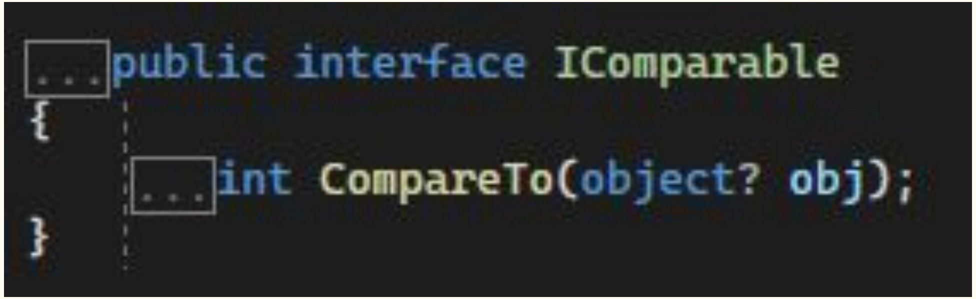
Default Sorting

- Array.Sort(xxx) - using the System.IComparable
- xxx.Sort() – xxx is a collection – using default comparer (System.IComparable)
  - String objects are lexicographically ordered
  - Date objects are chronologically ordered
  - Number and sub-classes are ordered numerically

Sort Order

- **IComparable<T>** interface
- **IComparer<T>** Interface

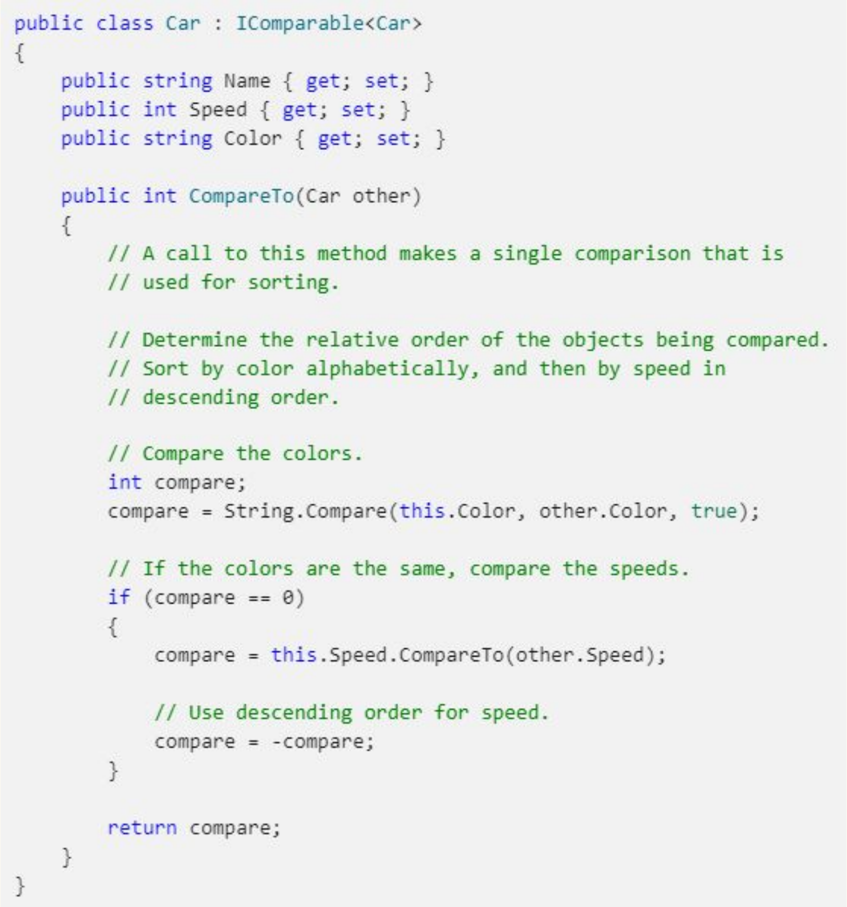
IComparable<T> interface



```
...public interface IComparable
{
    ...int CompareTo(object? obj);
}
```

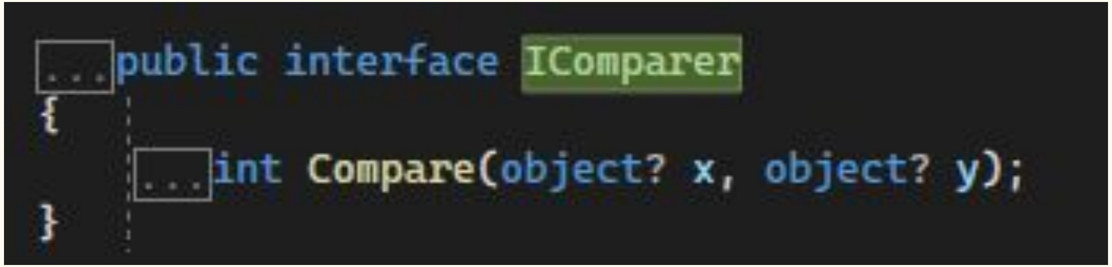
Compares the receiving object with the specified object

- Return value must be:
  - <0, if this precedes obj
  - ==0, if this has the same order as obj
  - >0, if this follows ob



```
public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}
```

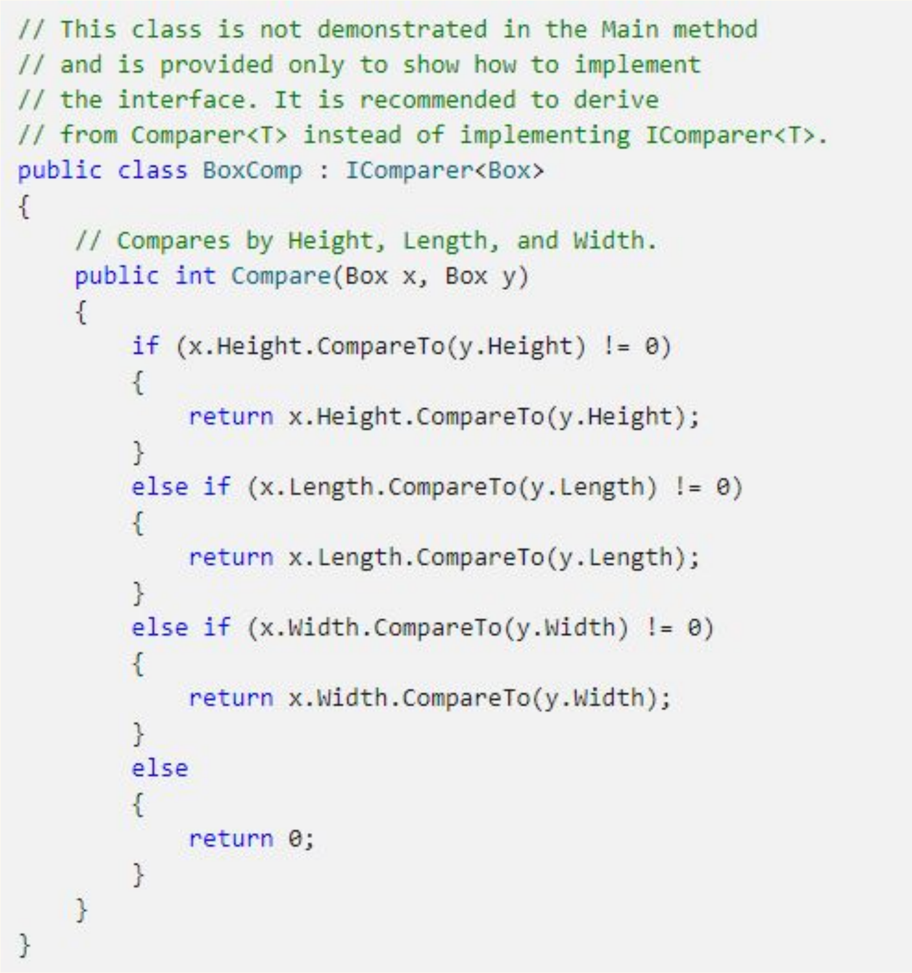
IComparer<T> Interface



```
...public interface IComparer
{
    ...int Compare(object? x, object? y);
}
```

Compares its two arguments

- Return value must be
  - <0, if x precedes y
  - ==0, if x has the same ordering as y
  - >0, if x follows y



```
// This class is not demonstrated in the Main method
// and is provided only to show how to implement
// the interface. It is recommended to derive
// from Comparer<T> instead of implementing IComparer<T>.
public class BoxComp : IComparer<Box>
{
    // Compares by Height, Length, and Width.
    public int Compare(Box x, Box y)
    {
        if (x.Height.CompareTo(y.Height) != 0)
        {
            return x.Height.CompareTo(y.Height);
        }
        else if (x.Length.CompareTo(y.Length) != 0)
        {
            return x.Length.CompareTo(y.Length);
        }
        else if (x.Width.CompareTo(y.Width) != 0)
        {
            return x.Width.CompareTo(y.Width);
        }
        else
        {
            return 0;
        }
    }
}
```

Questions?